Universidad Nacional de Córdoba

Facultad de Ciencias Exactas Físicas y Naturales

Informe Trabajo Práctico Final

Sampler de Sonido con LPC1769

Estudiantes:

Cabrera, Augusto Gabriel
Gil Cernich, Manuel
Schroder Ferrando, Florencia
Moroz, Esteban Mauricio

Profesores:

Ing Sanchez
Ing Gallardo

Electrónica Digital III

Córdoba,
16 de noviembre de 2023

Índice

1. Resumen.

Nuestro proyecto es crear un sampler de sonido que sirva como un medio para implementar conceptos acerca de la placa LPC1769 y otros módulos. Nuestro equipo esta formado enteramente por estudiantes de ingeniería en computación.

2. Enunciado.

Cuando se pulsa el botón `'Record'` conectado a `pin P2.10` , se captura la señal detectada por el micrófono conectado a `pin P0.22`. Esta señal analógica se almacena directamente en la memoria, sin necesidad del uso de DMA, siempre que se presione el botón. La zona de memoria se restablece al grabar un nuevo sonido. Además, un potenciómetro conectado al `pin P0.24` se utiliza para ajustar la frecuencia del sonido a través del `timeout` del DAC.

Cuando se presiona el botón `'play/stop'` conectado al `pin P2.11`, los datos se extraen de la memoria y se envían al DAC mediante DMA. Con el valor de la frecuencia dado por el potenciómetro en `pin P0.24`, se reproduce la señal de audio en el `pin P0.26` de salida analógica. Como método adicional de comunicación, se puede enviar una secuencia de sonidos en forma de valores decimales a través del `UART`, los cuales se guardan en la misma sección de memoria que la grabación por micrófono.

Lo anterior mencionado se puede diagramar como:

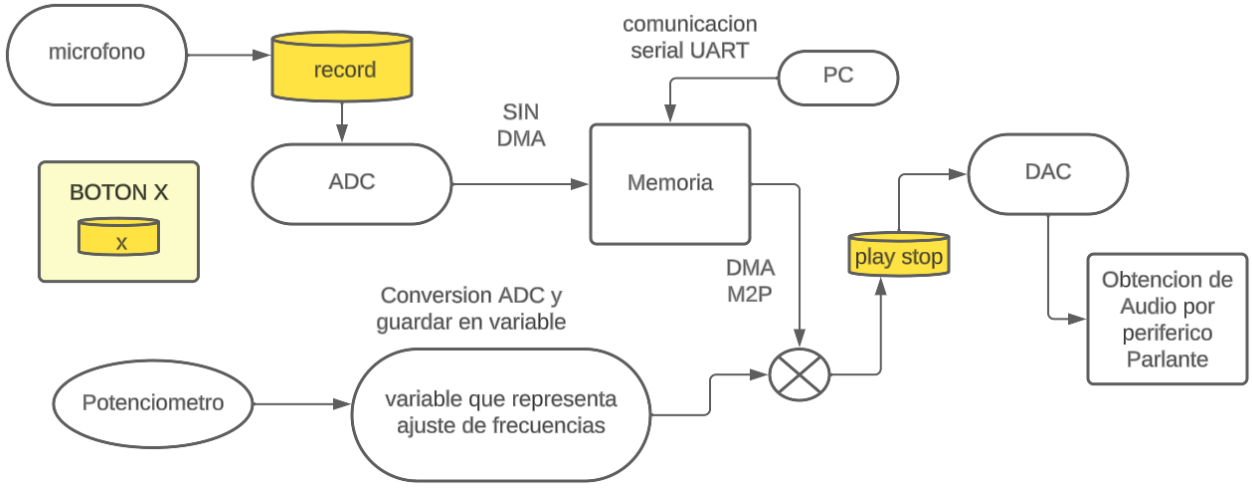

Figura 1: Diagrama a desarrollar

3. Marco Teórico.

3.1. Placa LPC1769

El LPC1769 es una Cortex ®Microcontrolador -M3 para aplicaciones integradas con un alto nivel de integración y bajo consumo de energía a frecuencias de 120 MHz. Las características incluyen:

- 512 kB de memoria flash,
- 64 kB de memoria de datos,
- Ethernet MAC, dispositivo / host / OTG USB, controlador DMA de 8 canales,

- 4 UART, 2 canales CAN, 3 SSP / SPI, 3 I2C, I2S,
- ADC de 8 canales y 12 bits, DAC de 10 bits, control de motor PWM,
- 4 temporizadores de uso general, reloj de tiempo real de potencia ultrabaja con suministro de batería separado

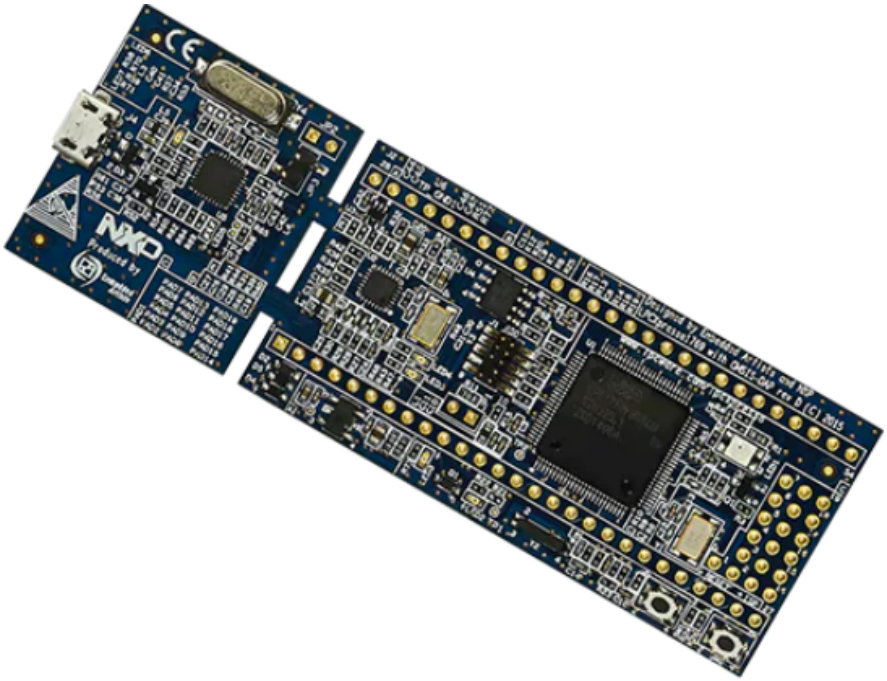

Figura 2: LPC1769 Cortex®-M3 microcontroller

3.2. Codificar MP3 a Decimal

A través del software "Audacity" ingresando un fragmento de audio en formato MP3, y con la ayuda del software "Encoder" se puede conseguir la codificación decimal del audio de interés.

Figura 3: App Audacity.

Figura 4: App EncodeAudio.

Los pasos a seguir son los siguientes:

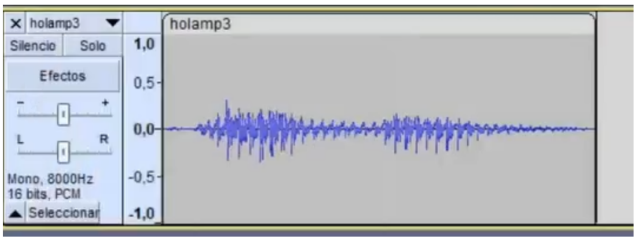

Figura 5: (PASO I) Abrir el archivo de audio (“holamp3.mp3”) formato MP3 en el software Audacity

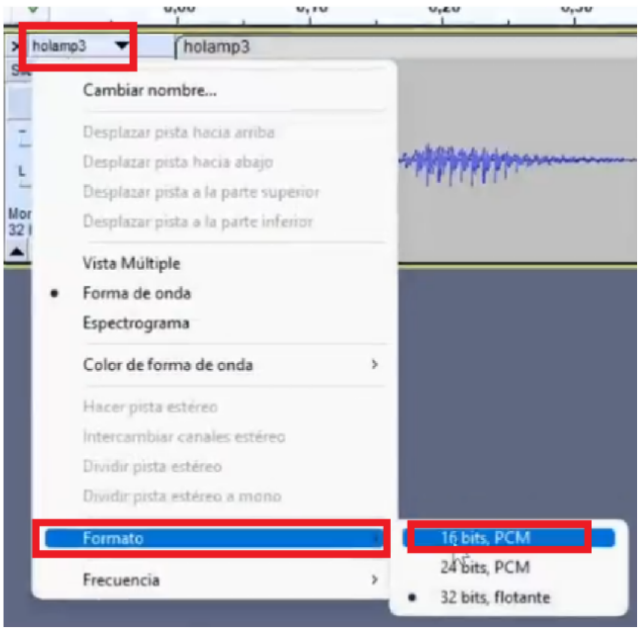

Figura 6: (PASO II) Seleccionar el formato de 16 bits PCM,

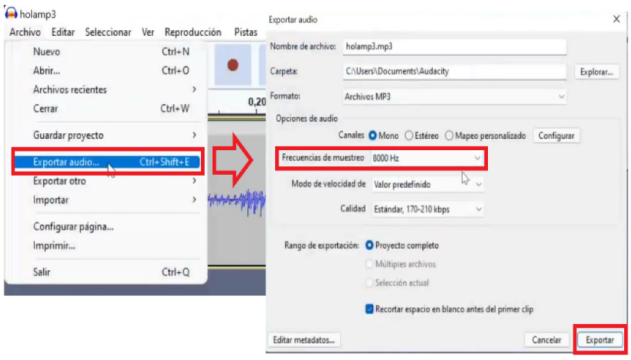

Figura 7: (PASO III) Para exportar el audio, se elige la mínima frecuencia de muestreo posible, esto afecta la calidad de audio, pero permite almacenar mayor cantidad de información decimal en la placa, este es el precio que pagar por usar un sistema con recursos limitados

Figura 8: (PASO IV) Busco el audio ("holamp3.mp3") con la app Encoder

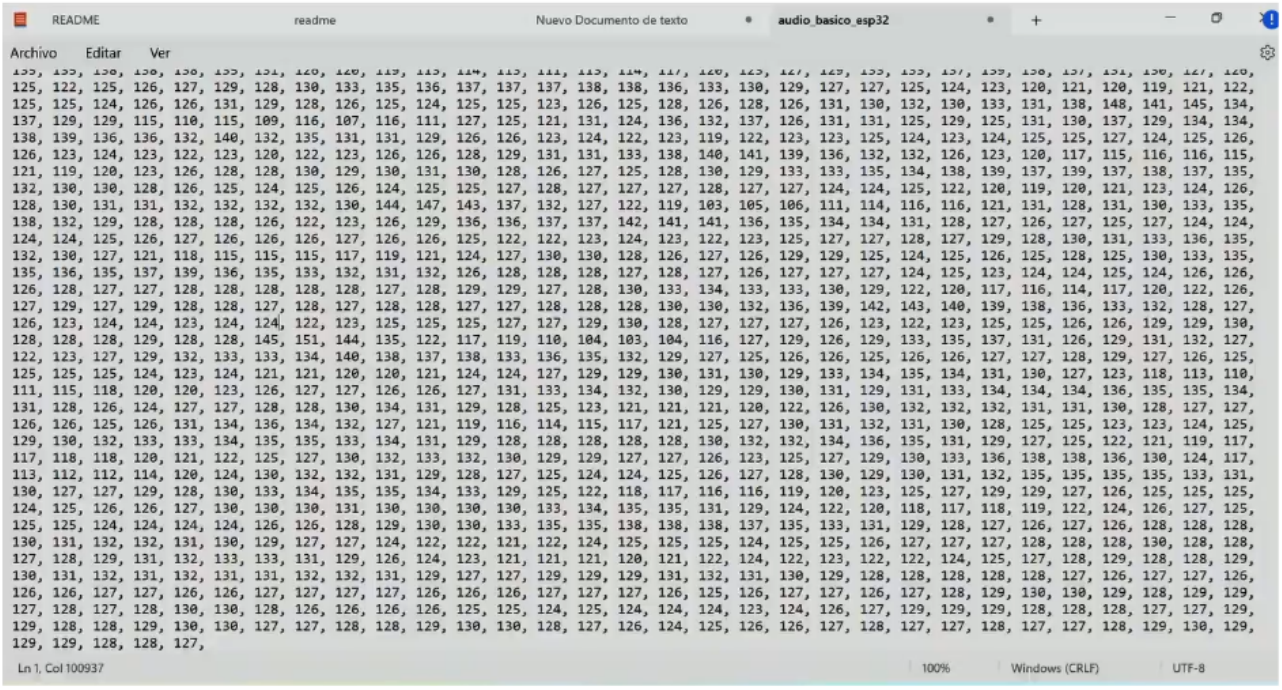

Figura 9: (PASO V) se guarda en el bloc de notas. Obteniendo la codificación a decimal.

3.3. Transformaciones de la Señal de audio entrante

La entrada inicial consiste en una señal analógica captada por un micrófono (Módulo Ky-037). Esta señal se convierte en su equivalente de aproximaciones sucesivas mediante el convertidor analógico a digital (ADC) de la placa, resultando en una representación digital. Esta última se almacena directamente en la memoria. Posteriormente, a través del uso del DMA (Direct Memory Access, controlador que se describe en la sección siguiente) para evitar interferir con los procesos críticos del procesador, los datos digitales se transfieren al módulo DAC para su reproducción final a través de un altavoz. La señal resultante es una representación bastante fiel de la señal original introducida en la placa. Es importante tener en cuenta que ambos convertidores tienen un número limitado de bits de conversión, lo cual puede afectar la claridad de las señales de audio. Sin embargo, este hecho no repercute en la comprensión de la misma.

El siguiente esquemático representa la transformación de la señal a través de la información

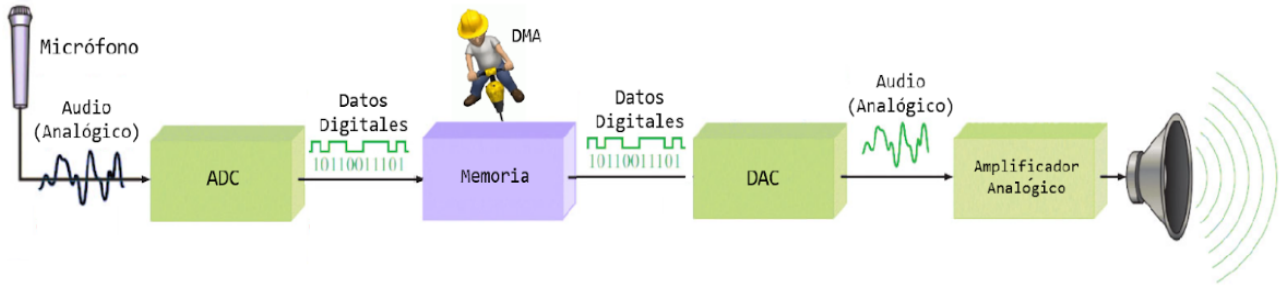

Figura 10: Esquema de transformación de la señal de audio analógica

El esquemático a continuación representa el accionar del DMA en el trabajo

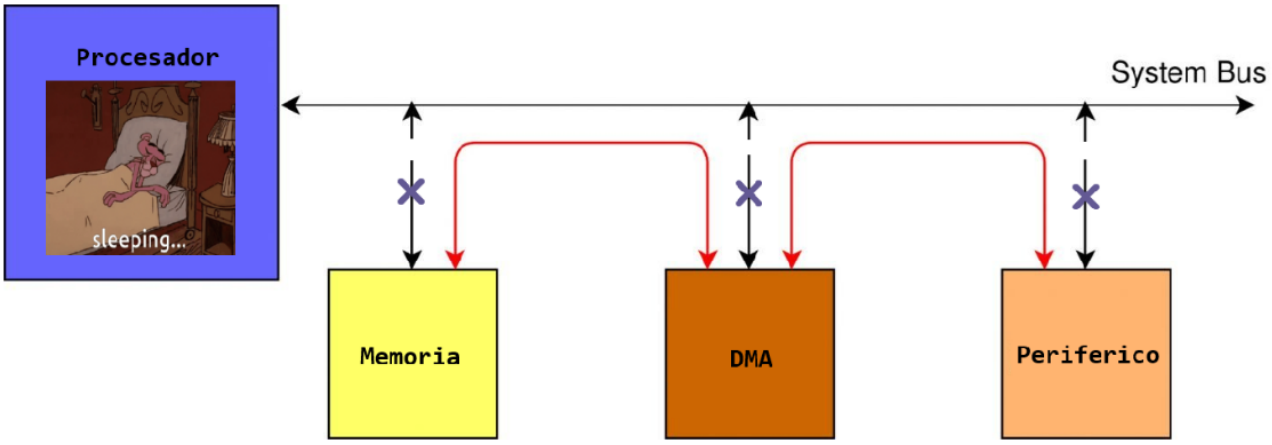

Figura 11: Esquema de trabajo del DMA

Básicamente, el DMA funciona mediante un controlador DMA que se encarga de gestionar las transferencias de datos entre los dispositivos periféricos y la memoria. Cuando se inicia una transferencia de datos, el controlador DMA toma el control del bus de datos y realiza la transferencia directamente entre el dispositivo periférico y la memoria, sin la intervención de la CPU. Esto funciona sin intervención

alguna del procesador, dándole tiempo al mismo para dedicarse a otras tareas mas relevantes.

4. Desarrollo

4.1. Componentes

A continuación se enumeran los componentes utilizados:

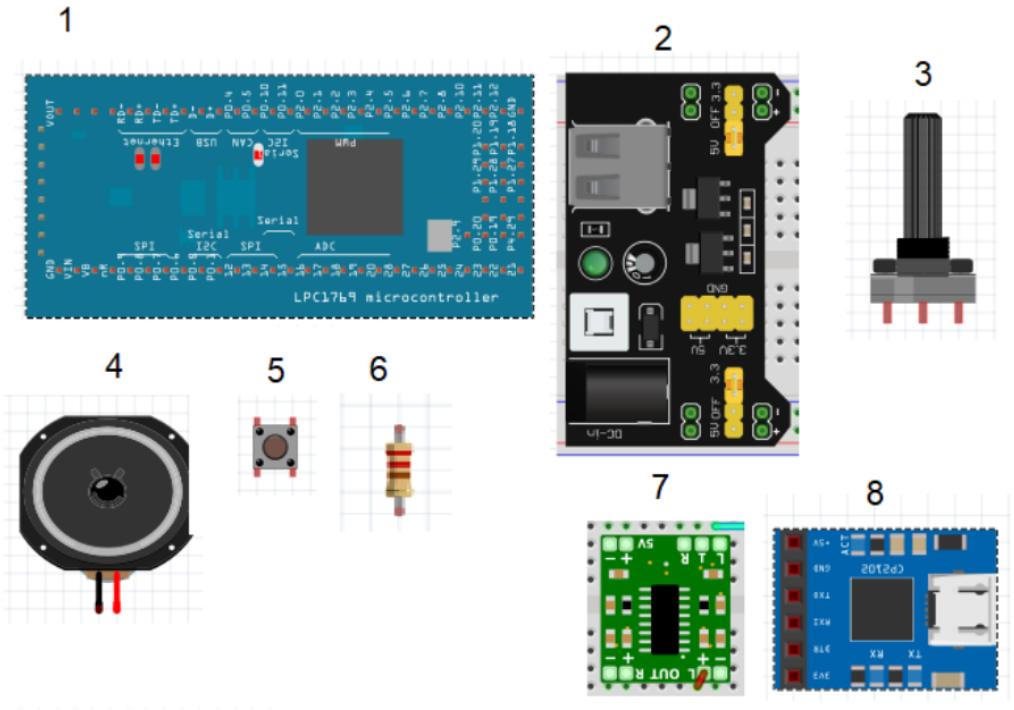

Figura 12: Componentes empleados

Se nombran a continuación:

1. LPC1769
2. Fuente POW-BREADBOARD 9v a 5v
3. Dos Potenciómetros: uno varía la frecuencia de 1 KHz y otro controla el volumen del parlante de 10 kHz
4. Parlante (8 ohm 1 watt)
5. Pulsador
6. Resistencias 10 KΩ
7. Amplificador PAM8403
8. TTL-UART Cp2102

4.2. Montado del circuito

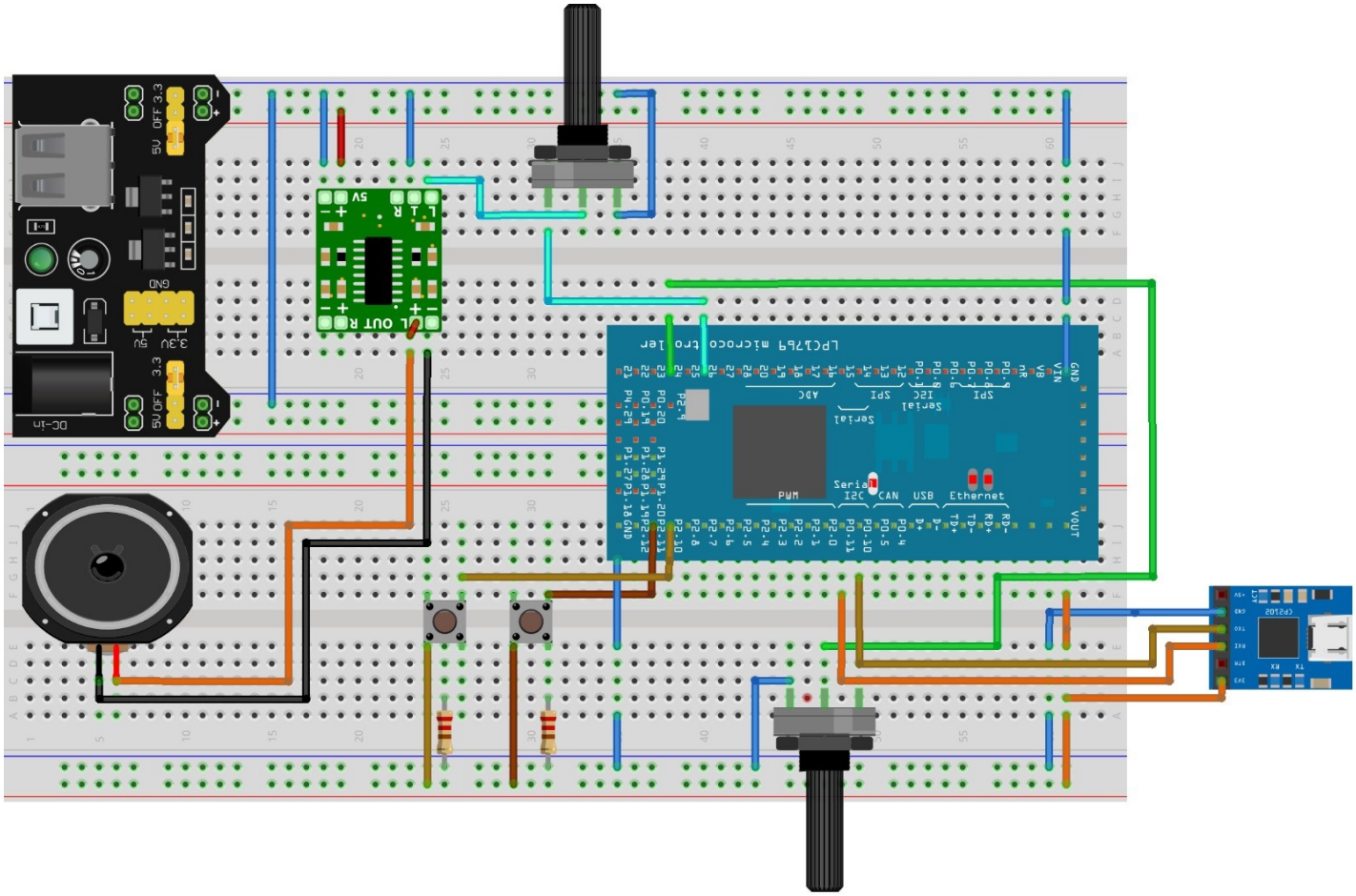

Figura 13: Montado circuital del proyecto

4.3. Cálculos

- **Cantidad de datos a guardar**: La capacidad máxima de la SRAM son 32 KB. En nuestro circuito cuando el micrófono graba los datos ya convertidos a digital se guardan en un arreglo `listADC` que almacena hasta 12000 datos `uint16_t` en la SRAM. 12000 datos ocupan 24 KB. En cambio, si almacenamos los datos en una variable declarada como constante le estaremos indicando al compilador que son variables `read-only`. De este modo, podremos almacenar más datos.

- **TimeOut**: El TimeOut es el valor en el cual interrumpe el controlador de DMA y saca las muestras por el pin `AOUT` del DAC que en nuestro circuito está conectado a un parlante.

  La fórmula para el `TimeOut` es:

$$\text{TimeOut} = \frac{\text{Fcclk}}{\text{Fmuestreo} \times N}$$

De esa forma variamos con un potenciómetro el `TimeOut`.

4.4. UART

Este trabajo práctico utiliza la comunicación serial UART en dirección `PC->LPC`.

Para la transmisión por UART primero se convierte un archivo MP3/WAV a valores decimales con "Audacity" y .[En]coderAudio". Estos valores se envían con un script de python que parsea los mismos y los envía de a UNO a la vez.

Para la configuración del UART se utilizó en su mayoría la configuración por defecto de los drivers de CMSIS utilizados en la cursada. Como por ejemplo un `baud-rate 9600 [bps]` más una configuración de FIFO con DMA deshabilitado, es decir que se interrumpe cada vez que se recibe un dato, y reseteo del buffer transmisión y recepción.

Las banderas de error que utilizamos en el handler son:

- UART_LSR _OE : Overrun error, un nuevo caracter se incluyo en una FIFO llena.
- UART_LSR _PE : Parity error.
- UART_LSR _FE : Framing error. Cuando el bit de stop es un cero.
- UART_LSR _BI : Break interrupt, Cuando el caracter completo de la transmisión es todo cero.
- UART_LSR _RXFE: Cualquier de los anteriores errores para la FIFO de Rx.

Como primera instancia en el handler testeamos si alguna de estas banderas es 1, si alguna está activada se queda en un loop infinito a modo de error, si no se toma el valor.

```
void configUART(void)
{
    UART_CFG_Type   UARTConfigStruct;
    UART_FIFO_CFG_Type UARTFIFOConfigStruct;

    // Configuracion por defecto 9600 baud-rate.
    UART_ConfigStructInit(&UARTConfigStruct);

    // Inicializa periferico
    UART_Init(LPC_UART2, &UARTConfigStruct);

    // Inicializa FIFO
    UART_FIFOConfigStructInit(&UARTFIFOConfigStruct);
    UART_FIFOConfig(LPC_UART2, &UARTFIFOConfigStruct);

    // Habilita interrupcion por el RX del UART
    UART_IntConfig(LPC_UART2, UART_INTCFG_RBR, ENABLE);
    // Habilita interrupcion por el estado de la linea UART
    UART_IntConfig(LPC_UART2, UART_INTCFG_RLS, ENABLE);
}
```

```
void UART2_IRQHandler(void)
{
    uint32_t intsrc, tmp, tmp1;
```

```
4
5       // Determina la fuente de interrupcion
6       intsrc = UART_GetIntId(LPC_UART2);
7       tmp = intsrc & UART_IIR_INTID_MASK;
8
9       // Evalua Line Status
10      if (tmp == UART_IIR_INTID_RLS)
11      {
12       tmp1 = UART_GetLineStatus(LPC_UART2);
13       tmp1 &= (UART_LSR_OE | UART_LSR_PE | UART_LSR_FE | UART_LSR_BI |
            UART_LSR_RXFE);
14       if (tmp1)
15       {
16           while(1){};      /* ingresa a un loop infinito si hay error */
17       }
18      }
19
20      // Receive Data Available or Character time-out
21      if ((tmp == UART_IIR_INTID_RDA) || (tmp == UART_IIR_INTID_CTI))
22      {
23       UART_Receive(LPC_UART2, info, sizeof(info), NONE_BLOCKING);
24      }
25
26  // A veces el UART tiene un bug que manda 0 de por medio por eso el if
       .
27      if ((count_UART < LISTSIZE) & (info[0] != 0))
28      {
29       listADC[count_UART] = (info[0]<<6);
30       count_UART++;
31      }
32
33      if (count_UART >= LISTSIZE)
34      {
35       count_UART = 0;
36      }
37
38      return;
39  }
```

5. Código

Nota: Se ha configurado el microcontrolador LPC1769 mediante el empleo de drivers. Estos consisten en un conjunto de funciones diseñadas específicamente para un hardware particular, en este caso, los drivers CMSIS están destinados para LPC1769/68. Es importante destacar que en otros tipos de placas de la misma familia, este conjunto de funciones controladoras no resulta funcional. Los drivers hacen uso de CMSIS para la generación de código; es crucial comprender que no se trata de entidades separadas, sino más bien de elementos interrelacionados.

```c
#include "LPC17xx.h"
#include "lpc_types.h"
#include "lpc17xx_adc.h"
#include "lpc17xx_dac.h"
#include "lpc17xx_gpdma.h"
#include "lpc17xx_pinsel.h"
#include "lpc17xx_exti.h"
#include "lpc17xx_uart.h"

// mayor ADCRATE mayor fidelidad  de sonido.
#define ADC_RATE    8000

// No superar los 15k muestras por que se llena la SRAM 32kB.
#define LISTSIZE    12000

// Arranca por default en un valor,  pero el potenciometro lo varia
// durante la ejecucion.
#define TIMEOUT     7000

#define NUM_LISTS   3           // Cada lista es de 4095 valores.

void configADC(void);
void configDAC(void);
void configGPIO(void);
void configEINT0(void);
void configUART(void);
void configEINT1(void);
void configDMA(__IO uint16_t listADC[]);
void configNVIC(void);

uint32_t map(uint32_t x, uint32_t in_min, uint32_t in_max, uint32_t
   out_min, uint32_t out_max);
void cleanListADC(void);
void moveListDAC(void);
void buttonDebounce(void);

/* Esta lista guarda el sonido grabado por el microfono. */
__IO uint16_t listADC[LISTSIZE] = {0};
__IO uint32_t *samples_count = (__IO uint32_t *)0x2007C000;
    // Contador de muestras para la lista del ADC.

/* Variables para la comunicacion UART. */
uint8_t info[1]     = "";
uint32_t count_UART = 0;

/* Var global para switchear de canal del ADC entre mic y potenc */
uint8_t RECORDING   = 0;

GPDMA_LLI_Type LLI_Array[NUM_LISTS];
```

```
49 GPDMA_Channel_CFG_Type dmaCFG;
50
51 /*----------------------------MAIN-------------------------------------*/
52
53 int main()
54 {
55     configGPIO();
56     configEINT0();
57     configEINT1();
58     configADC();
59     configDAC();
60     configUART();
61     configNVIC();
62
63     while(1)
64     {
65         // idle...
66     }
67
68     return 0;
69 }
70
71 /*----------------------------FUNCTIONS--------------------------------*/
72
73
74 void cleanListADC(void)
75 {
76     /* Rellenar con ceros la lista del ADC. */
77
78     for (uint32_t i = 0; i < LISTSIZE; i++)
79     {
80         listADC[i] = 0;
81     }
82 }
83
84
85 void moveListDAC(void)
86 {
87     /* Desplazamos los valor de la lista 6 lugares, 4 para el DAC y 2
           mas para recortar los LSB. */
88
89     for (uint32_t i = 0; i < LISTSIZE; i++)
90     {
91         listADC[i] = listADC[i]<<6;
92     }
93     return;
94 }
95
96
```

```
97  void buttonDebounce(void)
98  {
99      /* Delay para antirrebote del botones.
100       Se deberia hacer con un TIMER, no de esta manera. */
101
102     for (uint32_t i = 0; i < 50000; i++){}
103 }
104
105
106 uint32_t map(uint32_t x, uint32_t in_min, uint32_t in_max, uint32_t
        out_min, uint32_t out_max)
107 {
108     /* Convierte el valor recibido a un valor correspondiente dentro
109         de una escala MIN-MAX dada. */
110
111 return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min
        ;
112 }
113
114 /*-----------------------------CONFIGS-------------------------------*/
115
116 void configGPIO(void)
117 {
118   /* Set P0.23 AD0.0 */
119     PINSEL_CFG_Type pinCFG;
120     pinCFG.Funcnum      = PINSEL_FUNC_1;
121     pinCFG.OpenDrain    = PINSEL_PINMODE_NORMAL;
122     pinCFG.Pinmode      = PINSEL_PINMODE_TRISTATE;
123     pinCFG.Pinnum       = PINSEL_PIN_23;
124     pinCFG.Portnum      = PINSEL_PORT_0;
125     PINSEL_ConfigPin(&pinCFG);
126
127   /* Set P0.24 AD0.1 */
128     pinCFG.Funcnum      = PINSEL_FUNC_1;
129     pinCFG.OpenDrain    = PINSEL_PINMODE_NORMAL;
130     pinCFG.Pinmode      = PINSEL_PINMODE_TRISTATE;
131     pinCFG.Pinnum       = PINSEL_PIN_24;
132     pinCFG.Portnum      = PINSEL_PORT_0;
133     PINSEL_ConfigPin(&pinCFG);
134
135   /* Set P2.10 EINT0*/
136     pinCFG.Funcnum      = PINSEL_FUNC_1;
137     pinCFG.OpenDrain    = PINSEL_PINMODE_NORMAL;
138     pinCFG.Pinmode      = PINSEL_PINMODE_PULLDOWN;
139     pinCFG.Pinnum       = PINSEL_PIN_10;
140     pinCFG.Portnum      = PINSEL_PORT_2;
141     PINSEL_ConfigPin(&pinCFG);
142
143   /* Set P2.11 EINT1 */
```

```
144     pinCFG.Funcnum      = PINSEL_FUNC_1;
145     pinCFG.OpenDrain    = PINSEL_PINMODE_NORMAL;
146     pinCFG.Pinmode      = PINSEL_PINMODE_PULLDOWN;
147     pinCFG.Pinnum       = PINSEL_PIN_11;
148     pinCFG.Portnum      = PINSEL_PORT_2;
149     PINSEL_ConfigPin(&pinCFG);
150
151   /* Set P0.26 AD0.0 */
152     pinCFG.Funcnum      = PINSEL_FUNC_2;
153     pinCFG.OpenDrain    = PINSEL_PINMODE_NORMAL;
154     pinCFG.Pinmode      = PINSEL_PINMODE_TRISTATE;
155     pinCFG.Pinnum       = PINSEL_PIN_26;
156     pinCFG.Portnum      = PINSEL_PORT_0;
157     PINSEL_ConfigPin(&pinCFG);
158
159   /* P.022 Output LED */
160     LPC_GPIO0->FIODIR   |= (1<<22);     // Led Rojo
161     LPC_GPIO3->FIODIR   |= (1<<25);     // Led Verde
162     LPC_GPIO3->FIODIR   |= (1<<26);     // Led Azul
163     LPC_GPIO0->FIOSET   |= (1<<22);     // Apaga el led rojo.
164     LPC_GPIO3->FIOSET   |= (1<<25);     // Apaga el led verde.
165     LPC_GPIO3->FIOSET   |= (1<<26);     // Apaga el led azul.
166
167   /* Configuracion pin de Tx y Rx */
168     pinCFG.Funcnum      = 1;
169     pinCFG.OpenDrain    = 0;
170     pinCFG.Pinmode      = 0;
171     pinCFG.Pinnum       = 10;
172     pinCFG.Portnum      = 0;
173     PINSEL_ConfigPin(&pinCFG);  // Tx
174     pinCFG.Pinnum       = 11;
175     PINSEL_ConfigPin(&pinCFG);  // Rx
176     return;
177 }
178
179
180 void configADC(void)
181 {
182   /* ADC se utiliza en modo burst para tener el maximo de resolucion.
183    * La interrupcion se activa en configNVIC() y comienza apagada ya
184    * que se prende con el pulsador en EINT0.
185    * Tenemos dos canales de interrupcion, uno para el mic y otro para
186    * el potenciometro, pero solamente activamos la interrupcion del
187    * microfono, ya que el potenciometro va a depender del estado de
188    * la variable RECORDING.
189    */
190
191     ADC_Init(LPC_ADC, ADC_RATE);
192     ADC_StartCmd(LPC_ADC, ADC_START_CONTINUOUS);
```

```
193     ADC_ChannelCmd(LPC_ADC, 0, ENABLE);
194     ADC_ChannelCmd(LPC_ADC, 1, ENABLE);
195     ADC_BurstCmd(LPC_ADC, ENABLE);
196     ADC_IntConfig(LPC_ADC, ADC_ADINTEN0, ENABLE);
197 }
198
199
200 void configDAC(void)
201 {
202     DAC_CONVERTER_CFG_Type dacCFG;
203     dacCFG.CNT_ENA = SET;
204     dacCFG.DMA_ENA = SET;
205
206   /* REVISAR CALCULO DE TIMEOUT. */
207     DAC_SetDMATimeOut(LPC_DAC, TIMEOUT);
208     DAC_ConfigDAConverterControl(LPC_DAC, &dacCFG);
209     DAC_Init(LPC_DAC);
210 }
211
212
213 void configDMA(__IO uint16_t listADC[])
214 {
215     for (int i = 0; i < NUM_LISTS; i++)
216     {
217         LLI_Array[i].DstAddr = (uint32_t) &(LPC_DAC->DACR);
218         LLI_Array[i].SrcAddr = (uint32_t)(listADC + i * 4095);
219
220     if (i == (NUM_LISTS - 1))
221         { LLI_Array[i].NextLLI = (uint32_t)&LLI_Array[0]; }
222     else
223         { LLI_Array[i].NextLLI = (uint32_t)&LLI_Array[i + 1]; }
224
225         LLI_Array[i].Control = 4095
226                              | (1 << 18)    // source width 16 bit
227                              | (1 << 22)    // dest width = word 32 bits
228                              | (1 << 26)  ; // source increment
229     }
230
231     dmaCFG.ChannelNum            = 0;
232     dmaCFG.TransferSize          = 4095;
233     dmaCFG.TransferWidth         = 0;
234     dmaCFG.TransferType          = GPDMA_TRANSFERTYPE_M2P;
235     dmaCFG.SrcConn               = 0;
236     dmaCFG.DstConn               = GPDMA_CONN_DAC;
237     dmaCFG.SrcMemAddr            = (uint32_t) listADC;
238     dmaCFG.DstMemAddr            = 0;
239     dmaCFG.DMALLI                = (uint32_t) &LLI_Array[0];
240
241     GPDMA_Init();
```

```
242     GPDMA_Setup(&dmaCFG);
243     GPDMA_ChannelCmd(0, ENABLE);
244     return;
245 }
246 
247 
248 void configEINT0(void)
249 {
250     /* Interrupcion para inicializar la grabacion y el ADC.
251      * Las interrupciones se activan en configNVIC()    */
252 
253     EXTI_InitTypeDef exti;
254     exti.EXTI_Mode      = EXTI_MODE_EDGE_SENSITIVE;
255     exti.EXTI_polarity  = EXTI_POLARITY_HIGH_ACTIVE_OR_RISING_EDGE;
256     exti.EXTI_Line      = EXTI_EINT0;
257 
258     EXTI_Config(&exti);
259 }
260 
261 
262 void configEINT1(void)
263 {
264     /* Interrupcion para poner play/pausa el sonido.
265      * Las interrupciones se activan en configNVIC()    */
266 
267     EXTI_InitTypeDef exti;
268     exti.EXTI_Mode      = EXTI_MODE_EDGE_SENSITIVE;
269     exti.EXTI_polarity  = EXTI_POLARITY_HIGH_ACTIVE_OR_RISING_EDGE;
270     exti.EXTI_Line      = EXTI_EINT1;
271 
272     EXTI_Config(&exti);
273 }
274 
275 
276 void configNVIC(void)
277 {
278     LPC_ADC->ADGDR &= LPC_ADC->ADGDR;
279     NVIC_EnableIRQ(ADC_IRQn);
280 
281     EXTI_ClearEXTIFlag(EXTI_EINT0);
282     NVIC_EnableIRQ(EINT0_IRQn);
283 
284     EXTI_ClearEXTIFlag(EXTI_EINT1);
285     NVIC_EnableIRQ(EINT1_IRQn);
286 
287     NVIC_EnableIRQ(UART2_IRQn);
288 
289     GPDMA_ChannelCmd(0, DISABLE);
290 }
```

```
291
292 void configUART(void)
293 {
294             UART_CFG_Type       UARTConfigStruct;
295             UART_FIFO_CFG_Type  UARTFIFOConfigStruct;
296
297   // Configuracion por defecto 9600 baud-rate.
298             UART_ConfigStructInit(&UARTConfigStruct);
299
300   // Inicializa periferico
301             UART_Init(LPC_UART2, &UARTConfigStruct);
302
303   // Inicializa FIFO
304             UART_FIFOConfigStructInit(&UARTFIFOConfigStruct);
305             UART_FIFOConfig(LPC_UART2, &UARTFIFOConfigStruct);
306
307   // Habilita interrupcion por el RX del UART
308             UART_IntConfig(LPC_UART2, UART_INTCFG_RBR, ENABLE);
309
310   // Habilita interrupcion por el estado de la linea UART
311             UART_IntConfig(LPC_UART2, UART_INTCFG_RLS, ENABLE);
312 }
313
314 /*--------------------------HANDLERS-------------------------------*/
315
316 void ADC_IRQHandler(void)
317 {
318     static uint32_t ADCVAL      = 0;
319     static uint32_t ADCVALMAP   = 0;
320
321     /* Tomamos una muestra del ADC y la guardamos en el array sin
           superar el limite de muestras. */
322
323     if (RECORDING > 0)
324     {
325         if (*samples_count <= LISTSIZE)
326         {
327             /* Comenzamos a grabar un audio y guardarlo en el array. */
328             LPC_GPIO3->FIOCLR |= (1<<26);   // Prende el led azul.
329             listADC[*samples_count] = ((LPC_ADC->ADDR0)>>6) & 0x3FF;
330             (*samples_count)++;
331         }
332         else
333         {
334             LPC_GPIO0->FIOSET |= (1<<22);   // Apaga el led rojo.
335             LPC_GPIO3->FIOSET |= (1<<25);   // Apaga el led verde.
336             LPC_GPIO3->FIOSET |= (1<<26);   // Apaga el led azul.
337             *samples_count = 0;
338             RECORDING = 0;
```

```
339             moveListDAC();
340         }
341     }
342     else if (RECORDING == 0)
343     {
344       // Si NO estamos grabando variamos la frec de salida del DAC.
345 
346         ADCVAL      = ((LPC_ADC->ADDR1)>>6) & 0x3FF;
347         ADCVALMAP   = map(ADCVAL, 0, 1024, 5000, 20000);
348         DAC_SetDMATimeOut(LPC_DAC, ADCVALMAP);
349     }
350         LPC_ADC->ADGDR &= LPC_ADC->ADGDR;
351 }
352 
353 
354 void EINT0_IRQHandler(void)
355 {
356     /* Comenzamos a grabar un sonido por el ADC. Al llenarse el array
357         de valores se detiene automaticamente la grabacion a la espera
358         de poner en play el sonido. */
359 
360     buttonDebounce();
361 
362     RECORDING = 1;
363 
364     LPC_GPIO0->FIOSET |= (1<<22);   // Apaga el led rojo.
365     LPC_GPIO3->FIOSET |= (1<<25);   // Apaga el led verde.
366     LPC_GPIO3->FIOSET |= (1<<26);   // Apaga el led azul.
367     GPDMA_ChannelCmd(0, DISABLE);
368     *samples_count = 0;
369     cleanListADC();
370 
371     NVIC_EnableIRQ(ADC_IRQn);
372     EXTI_ClearEXTIFlag(EXTI_EINT0);
373 }
374 
375 void EINT1_IRQHandler(void)
376 {
377    /* Si esta reproduciendo sonido, deshabilita el canal y baja a 0 la
378    salida. Si esta en pausa, pone en play la reproduccion de sonido. */
379             static uint8_t PLAY = 0;
380 
381     buttonDebounce();
382 
383     if (PLAY > 0)
384     {
385         LPC_GPIO3->FIOSET |= (1<<25);   // Apaga el led verde.
386         LPC_GPIO3->FIOSET |= (1<<26);   // Apaga el led azul.
387         LPC_GPIO0->FIOCLR |= (1<<22);   // Prende el led rojo.
```

```
388         GPDMA_ChannelCmd(0, DISABLE);
389         DAC_UpdateValue(LPC_DAC, 0);
390         PLAY = 0;
391     }
392     else
393     {
394         LPC_GPIO0->FIOSET |= (1<<22);    // Apaga el led rojo.
395         LPC_GPIO3->FIOSET |= (1<<26);    // Apaga el led azul.
396         LPC_GPIO3->FIOCLR |= (1<<25);    // Prende el led verde.
397         configDMA(listADC);
398         PLAY = 1;
399     }
400
401     EXTI_ClearEXTIFlag(EXTI_EINT1);
402 }
403
404 void UART2_IRQHandler(void)
405 {
406     uint32_t intsrc, tmp, tmp1;
407
408         // Determina la fuente de interrupcion
409     intsrc = UART_GetIntId(LPC_UART2);
410     tmp = intsrc & UART_IIR_INTID_MASK;
411
412         // Evalua Line Status - Received-line status.
413     if (tmp == UART_IIR_INTID_RLS)
414     {
415         tmp1 = UART_GetLineStatus(LPC_UART2);
416         tmp1 &= (UART_LSR_OE | UART_LSR_PE | UART_LSR_FE | UART_LSR_BI
                | UART_LSR_RXFE);
417         if (tmp1)
418         {
419             while(1){}; /* ingresa a un loop infinito si hay error */
420         }
421     }
422
423         // Receive Data Available or Character time-out
424     if ((tmp == UART_IIR_INTID_RDA) || (tmp == UART_IIR_INTID_CTI))
425     {
426         UART_Receive(LPC_UART2, info, sizeof(info), NONE_BLOCKING);
427     }
428
429     /* A veces el UART tiene un bug que manda 0 de por medio por eso el
            condicional. */
430
431     if ((count_UART < LISTSIZE) & (info[0] != 0))
432     {
433         listADC[count_UART] = (info[0]<<6);
434         count_UART++;
```

```
435     }
436
437     if (count_UART >= LISTSIZE)
438     {
439         count_UART = 0;
440     }
441
442     return;
443 }
```

6. Agregados

6.1. Posibles mejoras

Algunas mejoras que se podrían realizar:

- **Mejoras estéticas:**
  - El circuito podría ir en una caja impresa en 3D para tener un diseño más intuitivo y atractivo.
- **Mejoras de software:**
  - Se podría agregar distintos efectos como eco, reverb, distorsión, etc en la voz.
  - Se podría enviar la señal digitalizada por UART a algún dispositivo.

6.2. Errores encontrados

Algunos errores que tuvimos a lo largo del trabajo integrador fueron

- **Problemas de memoria:** Nos dimos cuenta de la diferencia entre la memoria read-only y SRAM, si quisiéramos obtener un sonido más claro y de mayor calidad entonces deberíamos declarar el arreglo como constante pero de esa forma no se podrían guardar los datos que transforma el ADC.
- **Problemas de configuración:** A veces nos olvidábamos de habilitar NVIC, configurar los pines, las resistencias internas, etc. Estos errores por lo general nos pasaban cuando hacíamos muchos cambios en el código y se nos pasaba algún detalle por alto.
- **Problemas de generación de audio:** Para transformar un sonido de ".mp3" a decimal utilizábamos otra aplicación antes de usar "Audacity". Con esa aplicación no se escuchaba bien el sonido que reproducimos. Luego de debuggear, cambiar el código e ir a las consultas nos dimos cuenta que el problema era que se estaba generando erróneamente el script de hexadecimal. Una vez que comenzamos a utilizar "Audacity" se soluciono.
- Si se declara una linkedlist por DMA mayor al tamaño máximo que puede almacenar la lista del ADC, el sonido no se reproduce en loop, se reproduce una sola vez y no vuelve a empezar.
- Cuando estando en modo pausa, se vuelve al modo play los primeros segundos se escucha un sonido agudo al ejecutarse la instrucción de channel enable.

- Si se envían más valores por UART de los que permite la lista, se pueden llegar a sobreescribir los primeros valores ya que funciona como un buffer circular, y si llega a ocurrir esto para la próxima recepción por UART se comienza a guardar los valores desde este nuevo index que no es el 0, por lo que la lista queda "desfasada".

7. Enlaces Web

- **Link Repositorio GitHub (Clic Aquí)**
- **Link Vídeo del Proyecto (Clic Aquí)**

8. Conclusión

En conclusión, el sistema diseñado presenta una funcionalidad eficiente y versátil para la captura, almacenamiento y reproducción de señales de audio. La activación del botón 'Record' permite la adquisición directa de la señal analógica del micrófono, la cual se almacena en la memoria sin la necesidad de utilizar DMA, simplificando el proceso y optimizando los recursos. La gestión de la memoria se realiza de manera efectiva al resetearse al grabar un nuevo sonido, asegurando un uso eficiente del espacio de almacenamiento.

Por otro lado, la reproducción se inicia mediante el botón 'play/stop', desencadenando la extracción de datos de la memoria y su envío al DAC a través de DMA. La frecuencia del sonido se ajusta dinámicamente mediante un potenciómetro, proporcionando flexibilidad en la reproducción del audio. La salida analógica resultante se dirige al pin designado (P0.26), ofreciendo una interfaz clara y accesible para la reproducción de sonidos.

Además, el sistema incluye una funcionalidad adicional de comunicación a través del UART, permitiendo la transferencia de secuencias de sonidos en forma de valores decimales.

En conjunto, la integración de estas características proporciona un sistema completo y flexible para la captura, almacenamiento y reproducción de audio, abriendo posibilidades de comunicación adicional a través del UART.